

Programación Funcional Avanzada

Haskell Paralelo – MapReduce

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2015

El famoso *Map-Reduce*

- *Google MapReduce* – infraestructura para procesamiento de grandes volúmenes de datos en paralelo.
- *Map* es `map` – operar sobre un contenedor, preservando estructura.
- *Reduce* es `fold` – consolidar, típicamente para obtener un resultado sumariado, los resultados de varios contenedores.
- Nos conviene que ambos componentes puedan evaluarse en paralelo.
- Nos limitaremos a una arquitectura SMP – paralelismo implícito.



MapReduce Paralelo en Haskell

```
mapReduce :: Strategy b      -- Para el map
           -> (a -> b)       -- mapping
           -> Strategy c     -- Para el fold
           -> ([b] -> c)     -- folding
           -> [a]            -- Lista original
           -> c              -- Resultado

mapReduce mapS mapF redS redF data =
  mapRes 'pseq' redRes
  where mapRes = parMap mapS mapF data
        redRes = redF mapRes 'using' redS
```

Sólo nos queda usarla :-)



Andamiaje *MapReduce* en Haskell

O'Sullivan, Goerzen y Stewart – Adaptado a GHC 7.4

Procesar bitácoras (web, correo, *syslog*, ...)

- Archivos de texto – una línea por registro.
- Grandes – **muy** grandes en la vida real.
- *Data Mining* – ¿quién? ¿cuándo? ¿cuánto? ...
- Aplicando las lecciones aprendidas
 - ¿Cómo? – procesar archivos en paralelo
 - ¿Qué? – la función de exploración sobre *cada* línea.
 - ¿Cuánto? – unidad de trabajo por chispa (*chunking*).
 - Dominar la flojera ... del lenguaje.
 - ... más los problemas derivados de lidiar con el mundo exterior.



¿Cuánto?

- Trabajo en cada chispa mayor que el costo de administración.
- Procesar por partes – por línea es poco, usaremos *chunks* (“toletes”).

```
data ChunkSpec = CS {  
    chunkOffset :: !Int64,      -- Estricto!  
    chunkLength :: !Int64      -- Estricto!  
} deriving (Eq, Show)
```

- El constructor es estricto – **números** en lugar de *thunks*.
- Aprovecharemos Lazy ByteString en lugar de String – diseñados para lectura perezosa en bruto.
- Como el I/O tiene latencia, haremos la partición proporcional a la cantidad de *cores* disponibles.



Un breve compendio de manejo de excepciones

... porque haremos cosas con el mundo exterior

- Necesitamos un recurso, operar sobre él y liberarlo – pero garantizar la liberación si hay una excepción.

```
bracket :: IO a          -- obtener el recurso  
      -> (a -> IO b)    -- liberar el recurso  
      -> (a -> IO c)    -- operar sobre el recurso  
      -> IO c
```

- Típico para manejar archivos

```
bracket (openFile "foo.txt" ReadMode)  
      hclose  
      (\handle -> ...)
```



Variantes interesantes

No son más que bracket refinado

- Liberar sólo si ocurrió una excepción durante la operación

```
bracketOnError :: IO a  
              -> (a -> IO b)  
              -> (a -> IO c)  
              -> IO c
```

- Sólo interesan los efectos de borde

```
bracket_ :: IO a -> IO b -> IO c -> IO c
```

- Sin operación intermedia – garantiza la ejecución de la segunda.

```
finally :: IO a -> IO b -> IO a
```



Procesando *chunks* de un archivo

```
withChunks :: (NFData a) =>
    (FilePath -> IO [ChunkSpec])
    -> ([LB.ByteString] -> a)
    -> FilePath
    -> IO a

withChunks chunkFunc process path = do
    (chunks, handles) <- chunkedRead chunkFunc path
    let r = process chunks
    (runEval (rdeepseq r) 'pseq' return r)
    'finally' mapM_ hClose handles

chunkedReadWith :: (NFData a) =>
    ([LB.ByteString] -> a)
    -> FilePath -> IO a

chunkedReadWith func path =
    withChunks (lineChunks (numCapabilities * 4))
    func path
```


Procesando *chunks* de un archivo

```
chunkedRead :: (FilePath -> IO [ChunkSpec])
             -> FilePath
             -> IO ([LB.ByteString], [Handle])

chunkedRead chunkFunc path = do
  chunks <- chunkFunc path
  liftM unzip . forM chunks $ \spec -> do
    h <- openFile path ReadMode
    hSeek h AbsoluteSeek
          (fromIntegral (chunkOffset spec))
    chunk <- LB.take (chunkLength spec)
              'liftM' LB.hGetContents h
  return (chunk, h)
```

- `chunkFunc` determina el tamaño de cada *chunk*.
- Un *filehandle* por cada tolete – lectura paralela.
- Cada uno se posiciona y lee (¡con flojera!).

Procesando *chunks*

Primera parte – ubicación grosera

```
lineChunks :: Int -> FilePath -> IO [ChunkSpec]
lineChunks numChunks path = do
  bracket (openFile path ReadMode) hClose $ \h -> do
    totalSize <- fromIntegral 'liftM' hFileSize h
    let chunkSize = totalSize
                        'div' fromIntegral numChunks
    findChunks offset = do
      let newOffset = offset + chunkSize
      hSeek h AbsoluteSeek $ fromIntegral newOffset
      -- falta encontrar el '\n' cercano
    findChunks 0
```

- bracket “envuelve” el procesos de chunking entre apertura y cerrado del archivo – resistente a excepciones.
- Partes iguales según el tamaño del archivo.
- En cada parte hay que encontrar el salto de línea más cercano.



Procesando *chunks*

Segunda parte – ubicación fina

```
let findNewline off = do
  eof <- hIsEOF h
  if eof
  then return [CS offset (totalSize - offset)]
  else do
    bytes <- LB.hGet h 4096
    case LB.elemIndex '\n' bytes of
      Just n -> do
        chunks@(c:_) <- findChunks (off + n + 1)
        let coff = chunkOffset c
        return (CS offset (coff - offset):chunks)
      Nothing -> findNewline (off + LB.length bytes)
findNewline newOffset
```

- El último *chunk* será más pequeño – no afecta.

Listo el andamiaje...



El programa principal

- El andamiaje sabe procesar cualquier archivo en *chunks*.
- Nuestro mapReduce genérico sabe cómo hacerlo en paralelo.
- Falta indicar **qué** queremos calcular.



¿Cuántas líneas tiene el archivo?

```
lineCount :: [LB.ByteString] -> Int64
lineCount = mapReduce rdeepseq (LB.count '\n')
              rdeepseq sum

main :: IO ()
main = do
  args <- getArgs
  forM_ args $ \path -> do
    numLines <- chunkedReadWith lineCount path
    putStrLn $ path ++ ": " ++ show numLines
```

- Contar los saltos de línea en cada *chunk*.
- Sumar los conteos parciales.
- Ambas operaciones evaluadas *completamente* – no queremos *thunks* demorados hasta al final.



¿Cuántas líneas tiene el archivo?

Las corridas

- Un *core*

```
$ time ./count +RTS -N1 -RTS access.log.6  
access.log.6: 709753  
  
real 0m0.625s
```

- Dos *cores*

```
$ time ./count +RTS -N2 -RTS access.log.6  
access.log.6: 709753  
  
real 0m0.458s
```

Una mejora cercana al 30 %



¿Cuántas líneas tiene el archivo?

El análisis

```
$ time ./count +RTS -N2 -sstderr -RTS access.log.6
```

- 178,820 bytes maximum residency (7 sample(s))
- 4 MB total memory in use (1 MB lost due to fragmentation)
- Parallel GC work balance: 1.34 (26920 / 20073, ideal 2)
- SPARKS: 8 (7 converted, 1 pruned)
- Productivity 99.4 % of total user

Optimizar más sería *caro* – luce bien.



¿Cuáles son los 10 URLs más populares?

La función de conteo

- Data.Map para guardar los URLs.
- Regex.PCRE para filtrar con expresiones regulares.

```
countURLs :: [L.ByteString] -> M.Map S.ByteString Int
countURLs = mapReduce
    rseq (foldl' augment M.empty . L.lines)
    rseq (M.unionsWith (+))
where augment map line =
    case match (compile pattern []) (strict line) [] of
        Just (_,url:_) -> M.insertWith' (+) url 1 map
        _ -> map
strict = S.concat . L.toChunks
pattern = S.pack "\"(?:GET|POST|HEAD) ([^ ]+) HTTP/"
```



¿Cuáles son los 10 URLs más populares?

El programa principal

```
main = do
  args <- getArgs
  forM_ args $ \path -> do
    m <- chunkedReadWith countURLs path
    let mostPopular (_,a) (_,b) = compare b a
    mapM_ print . take 10 .
      sortBy mostPopular . M.toList $ m
```



¿Cuáles son los 10 URLs más populares?

Las corridas

- Un *core*

```
$ time ./urls +RTS -N1 -RTS access.log.6  
("/test.txt",258391)  
...  
  
real 0m13.697s
```

- Dos *cores*

```
$ time ./urls +RTS -N2 -RTS access.log.6  
("/test.txt",258391)  
...  
  
real 0m9.113s
```

Una mejora cercana al 35%



¿Cuáles son los 10 URLs más populares?

El análisis

```
$ time ./urls +RTS -N2 -sstderr -RTS access.log.6
```

- 148,620,984 bytes maximum residency (9 sample(s))
- 191 MB total memory in use (2 MB lost due to fragmentation)
- Parallel GC work balance: 1.24 (30912027 / 24936088, ideal 2)
- SPARKS: 8 (7 converted, 1 pruned)
- Productivity 82.8 % of total user

Faltan detalles sobre consumo de memoria – ¿será la flojera?



Otra prueba . . .

- No hay *sparks* desperdiciados – las tareas tienen tamaño razonable.
- Parece que pasamos mucho tiempo en el GC.
- Ejecutaremos con un *heap* inicial más grande para que el GC no tenga que comenzar a trabajar tan pronto
 - Regla general: tres veces la máxima residencia.
 - En la próxima clase hablaremos más.
- Una nueva corrida

```
$ time ./urls +RTS -N2 -H400M -RTS access.log.6  
...  
real 0m9.671s
```

- Parallel GC work balance: 1.28 (23802734 / 18636446, ideal 2)
- Productivity 77.0 % of total user



Consideraciones finales

- Control del paralelismo – par y pseq.
- Estrategia de evaluación – ¿cuán estricto es el cálculo de cada valor?
- Granularidad – que cada tarea “valga la pena” para promover el *spark* en hilo.
- Localidad – hilos que dependan exclusivamente de valores locales.

Nuestros problemas con *MapReduce* son diferentes:
el hardware de I/O no es suficientemente **rápido**



Parallel Monads

Control.Monad.Par

- Par – Monad para acelerar cálculos puros usando varios núcleos.
- Construido usando continuaciones.
- Functor y Applicative – porque sólo es para cálculos puros.
- Especificar cálculos paralelos puros en los cuales el orden de cálculo no se conoce de antemano, sólo el flujo de datos.
 - Programador especifica el flujo de datos usando IVars.
 - Monad decide el orden de evaluación según la cantidad de núcleos, pero siempre manteniendo las dependencias.
 - Par es de resultado determinístico, pero orden de cómputo no determinístico.



Paralelismo Monádico

Expresando flujo de datos simple

- Calcular $(f\ x)$ y $(g\ x)$ – producir tupla con los resultados

```
runPar $ do
  fx <- pval (f x)
  gx <- pval (g x)
  a  <- get fx
  b  <- get gx
  return (a,b)
```

- pval – inicia evaluación en paralelo esperando resultado en un `IOa1`
- get – esperar a que haya resultado en un `IOa1`



Expresar Paralelismo Monádico

IVar para sincronizar flujo de datos

- IVar – variables compartidas con estrategia de evaluación

```
new      :: Par (IVar a)
newFull  :: NFData a => a -> Par (IVar)
```

- get – retorna *una sola vez* cuando hay valor en el IVar

```
get :: IVar a -> Par a
```

- get – pone un valor en el IVar *una sola vez*.

```
put :: NFData a => IVar a -> a -> Par ()
```

- Más de un put a un mismo IVar – error a tiempo de ejecución.
- put evalúa completamente el valor – estricto para obligar a calcular.
- Variante put_ sólo llega hasta WHNF.



Operaciones Paralelismo Monádico

Indicar cálculos en paralelo

- `fork` – iniciar otra tarea en paralelo

```
fork :: Par () -> Par ()
```

- `spawn` – `fork` pero con un `IVar` asociado al `IVar`

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  r <- new
  fork $ p >>= put r
  return r
```

- `pval` – `spawn` pero proveyendo el valor puro.

```
pval :: NFData a => a -> Par (IVar a)
pval = spawn . return
```



De flujo de datos a paralelismo

- Sea un cómputo tal que a partir de a.
 - d depende de b y c.
 - c depende de a.
 - b depende de a.

```
runPar $ do
  [a,b,c,d] <- sequence $ replicate 4 new
  fork $ do x <- get a
             put b (x+3)
  fork $ do x <- get a
             put c (x+4)
  fork $ do x <- get b
             y <- get c
             put d (x*y)
  fork $ do put a (3 :: Int)
  get d
```

- El resultado siempre será 42.



Parallel Monad MapReduces

- `parMap` – paralelismo puro sobre cualquier `Traversable`

```
parMap :: (Traversable t, NFData b) =>  
         (a -> b) -> t a -> Par (t b)
```



Parallel Monad MapReduces

- `parMap` – paralelismo puro sobre cualquier Traversable

```
parMap :: (Traversable t, NFData b) =>
        (a -> b) -> t a -> Par (t b)
```

- `parMapReduceRangeThresh` – map/reduce sobre un rango finito

```
parMapReduceRangeThresh :: NFData a
    => Int                    -- umbral
    -> InclusiveRange        -- rango para calcular
    -> (Int -> Par a)        -- Map a operar
    -> (a -> a -> Par a)    -- Combinar
    -> a                     -- Valor inicial
    -> Par a
```

- Dividir el rango en dos – calcularlos en paralelo.
 - Si el rango es menor que el umbral, no partir más.
- `parMapReduceRange` – paricionamiento según núcleos.

Monad Paralelo en acción

- En lugar de...

```
ghci> foldl' (+) 0 (map (^2) [1..106])
```



Monad Paralelo en acción

- En lugar de...

```
ghci> foldl' (+) 0 (map (^2) [1..106])
```

- ...se puede escribir

```
ghci> runPar $ parMapReduceRangeThresh  
           100  
           (InclusiveRange 1 (106))  
           (\x -> return (x2))  
           (\x y -> return (x+y))  
           0
```

...y aprovechar núcleos es cuestión de -N



Quiero saber más...

- [Página de WikiPedia sobre MapReduce](#)
- [Documentación de `Control.Monad.Par`](#)
- [Documentación de `Control.Exception`](#)
- [Documentación de `System.IO`](#)

